Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

CONTENTS

FEATURES

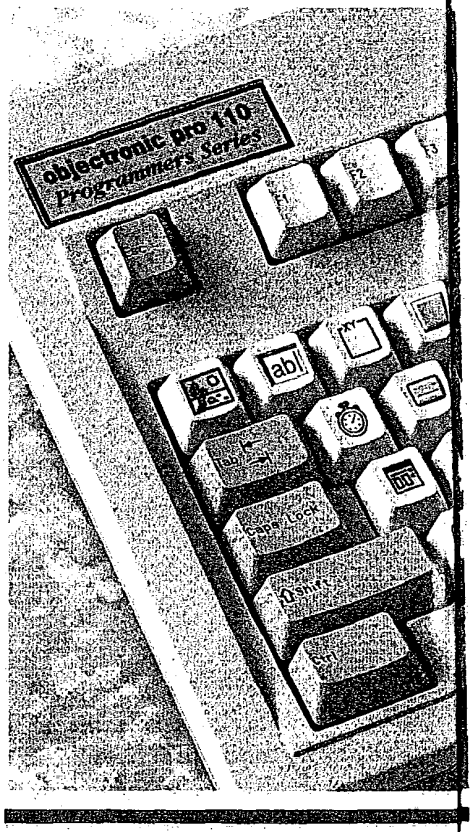
AUTOMATING ASSOCIATION IMPLEMENTATION IN C++ 18
by David M. Papurt
David compares several unidirectional and bidirectional pointer-based methods for implementing one-to-one associations: a direct, handwritten implementation, a modular approach exploiting inheritance, and template-based implementations.

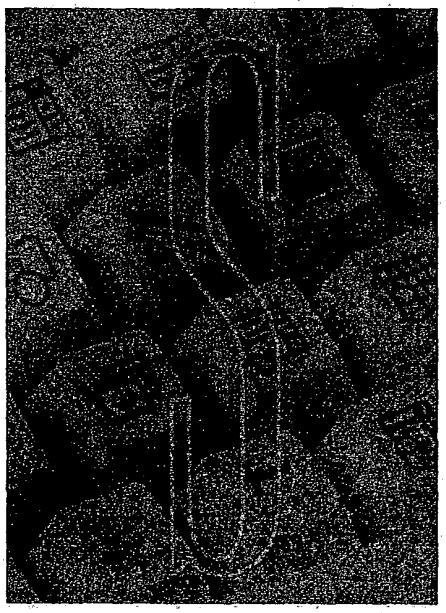
OBJECT-ORIENTED FACILITIES IN ADA 95 28
by David L. Moore
With the adoption of the ISO/ANSI Ada 95 standard, Ada supports object-oriented programming features such as class-wide objects, private types and child packages, multiple inheritance, and more.

PARTIAL REVELATION AND MODULA-3 36
by Steve Freeman
When compared to strongly typed languages, Modula-3 gives you greater flexibility in class reuse. Steve examines Modula-3's type system and describes how you can take advantage of its power.

OBJECT-ORIENTED PROGRAMMING IN S 44
by Richard Calaway
S, a high-level, object-oriented language, was originally designed for data analysis and graphics. As Richard points out, however, the S language is useful for a wide range of applications.

COBOL '97: A STATUS REPORT 52
by Henry Saade and Ann Wallace
The proposed Cobol '97 standard includes object-oriented features such as class definition, subclassing, data encapsulation, and polymorphism. Our authors focus on the object-oriented extensions to Cobol, and cover other proposed features.





FILE-STREAMING CLASSES IN C++ 58
by Kirit Saelensminde
Kirit implements a C++ file-streaming system that, unlike MFC or OWL, doesn't require a common superclass. This approach leads to less overhead and greater portability.

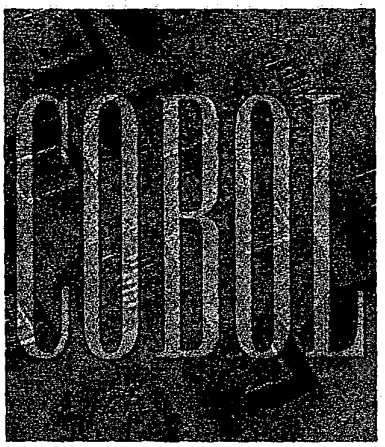
INSIDE MFC SERIALIZATION 62
by Jim Beveridge
The Microsoft Foundation Class Library implements a typesafe serialization mechanism that is both fast and flexible. Jim takes an in-depth look at how this mechanism works—and examines how you can get the most out of it.

EMBEDDED SYSTEMS

INSIDE FLASH MEMORY 68
by Brian L. Dipert
Direct-execute flash memory systems don't require the gigabyte hard disks and 64-Mbit DRAM arrays common in desktop systems.

E N T S

OCTOBER 1995
VOLUME 20, ISSUE 10



NETWORKED SYSTEMS

ENVIRONMENT VARIABLES AND WINDOWS 3.1 84
by John (Fritz) Lowrey
Nothing can give network administrators a headache like several hundred networked computers which need to run software that demands customized run-time environments. To make the job even more difficult, using a single environment space in the Windows system area for all programs is dangerous, not to mention that environment handling under Windows 3.1 is poorly documented.

EXAMINING ROOM

EXAMINING CA-VISUAL OBJECTS 90
by Rod da Silva
CA-Visual Objects is an application-development environment that sports an incremental, native-code compiler, visual painters and editors, an advanced, active, repository-based storage system, and an object-oriented language that allows for optional strong typing and full object orientation.

PROGRAMMER'S WORKBENCH

POWERBUILDER NVOs 103
by Mark Robinson
Quite possibly, Non-Visual User Objects (NVOs) are PowerBuilder's most useful tool for creating true object-oriented applications. Mark examines the effective use of NVOs and their role in application development.

COLUMNS

PROGRAMMING PARADIGMS 131
by Michael Swaine
Michael takes a look at Sun's HotJava Web browser, as well as Java, the programming language it was written in.

58 **C PROGRAMMING** 137
by Al Stevens
Al wraps up "MidiFitz," a Windows-based C++ program that uses MIDI to emulate a jazz piano player's rhythm section in real time. MidiFitz examines the notes being played, deduces a musical chord, and plays a bass line through the MIDI system.

62 **ALGORITHM ALLEY** 145
edited by Bruce Schneier
This month, Louis Plebani examines search procedures that enable you to obtain the common fraction approximation of real numbers. In doing so, he focuses on number theories such as Farey Sequences.

PROGRAMMER'S BOOKSHELF 149
by Ray Valdés
68 A good C++ book can be hard to find. Ray looks at some of the better ones.

FORUM

EDITORIAL 6
by Jonathan Erickson

LETTERS 10
by you

SWAINE'S FLAMES 168
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST 164
by Monica E. Berg

SOURCE CODE AVAILABILITY

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (California residents add sales tax) to *Dr. Dobb's Journal*, 411 Borel Ave., San Mateo, CA 94402, call 415-655-4100 x5701, or use your credit card to order by fax, 415-358-9749. Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ), via anonymous FTP from site ftp.mv.com (192.80.84.3) in the /pub/ddj directory, on the World Wide Web at http://www.ddj.com, and through DDJ Online, a free service accessible via direct dial at 415-358-8857 (1200/2400/9600 baud, 8-N-1).

NEXT MONTH

From the front cover to the back page, we'll be covering client/server architectures in our November issue.

Inside MFC Serialization

Typesafe serialization that's fast and flexible

Jim Beveridge

Having seen several commercial software packages through complete life cycles, I am reluctant to rely on "black-box" solutions, which tend to break down as a package evolves and becomes more complex. When I first saw the serialization mechanism in the Microsoft Foundation Classes (MFC), I questioned whether it was robust and flexible enough for a commercial application. I discovered that, although it has limitations, the serialization mechanism in MFC is strongly grounded in modern, object-oriented design theory. Furthermore, it is typesafe and leaves room for your design to evolve.

Using MFC serialization is straightforward. By default, any class derived from *CObject* can include a *Serialize()* member function that takes a *CArchive* as a parameter. In this member function, you add your own code to save and load any data associated with your class.

Data is serialized to and from a *CArchive* with *operator<<* and *operator>>*, much like *iostream*. The big difference is that *CArchive* is strictly a binary data format. As in *iostream*, there are default implementations that read and write fundamental data types such as *long* and *char*. The absence of the data type *int* facilitates portability between 16- and 32-bit implementations. The default implementations also handle byte swapping for types that support it. (For more information on portability between Little- and Big-endian architectures, see "Endian-Neutral Software," by James R. Gillig, *DDJ*, October/November 1994.)

MFC's implementation seemed obvious and uninteresting until the day I created multiple document types in the same application. At that point, I noticed that whenever I loaded a file, MFC would correctly create the right kind of document object and call the proper *Serialize()* member function. This happened in spite of the fact that I had not written any code to help MFC create these document types. Or so I thought....

Problems, Problems Everywhere

To create a document or any other kind of object on the fly, MFC needs to solve three problems:

Problem 1. Arbitrary types must be created as needed, but the *new* operator can only create an explicit type, so a form of "virtual constructors" for *CObject* is necessary.

Problem 2. Developers need to be able to easily add new classes to be created. Ideally, this would be done in the class definition and/or implementation.

Problem 3. A mapping scheme is needed to allow a particular type to be created based on information read from a file. This mapping cannot be hardcoded in MFC because developers add new types all the time.

As you'll see, MFC solves these problems elegantly with the use of a registry with automatic type registration and an implementation of virtual constructors based on these registered types. MFC's run-time type information is a building block for this architecture.

The Type Registry

To handle run-time type information, MFC creates a registry of classes in the application that are derived from *CObject*. This has nothing to do with the OLE registry, but the concept is similar. The type registry is a linked list of *CRuntimeClass* structures, in which each entry describes a *CObject*-derived class in the application. Listing One shows the *CRuntimeClass* structure (listings begin on page 122).

The real magic is that the types in this registry are not hardcoded in any table. The first clue to how this trick is accomplished is at the top of SCRIBDOC.H from the MFC "Scribble" sample application. The beginning of the class declaration looks like Example 1(a).

The online help says to use the DECLARE_DYNCREATE macro to enable objects of *CObject*-derived classes to be created dynamically at run time. Although this is a concise description of what DECLARE_DYNCREATE does, what really goes on inside the macro is far more interesting. After preprocessing, the DECLARE_DYNCREATE macro expands to several new class members; see Example 1(b). (Note that all examples are from MFC 3.1 and Visual C++ 2.1. I've reformatted all preprocessor-generated code for readability.)

The *GetRuntimeClass()* virtual function is the basis for run-time types in MFC. Run-time type information can be accessed for any *CObject*-derived object that includes DECLARE_DYNAMIC, DECLARE_DYNCREATE, or DECLARE_SERIAL. This information lets you determine if an object can legally be "downcast" to a derived

Jim, a software developer at Turning Point Software, can be contacted at jimb@turningpoint.com.

(continued from page 62)
class or if one object is the same class as another. Although Visual C++ does not support the new C++ operator *dynamic_cast*, using this run-time type information will achieve the same effect.

The run-time type information is declared using a static member variable, in this case *classCScribDoc*. The name (which has no space) is created in the various DECLARE_xxx macros with the macro-concatenation operator. Both *_GetBaseClass()* and *GetRuntimeClass()* are used to access this run-time class information. *GetRuntimeClass()* is virtual, so the type of an object can be found even with a pointer to a generic *CObject*.

Finally, the *Construct()* static member function forms the basis of MFC's use of its class registry as a class factory that can create arbitrary types on demand. To understand the workings of *Construct()*, some background is necessary.

Creating an Object

In *Advanced C++: Programming Styles and Idioms* (Addison-Wesley, 1992), James O. Coplien describes the concept of a virtual constructor:

> The virtual constructor is used when the type of an object needs to be determined from the context in which the object is constructed.

In MFC, the context is based on information read from a serialized archive. However, a virtual constructor is only a concept; no language construct implements it directly. The *new* operator requires an explicit class as its argument. Virtual constructors can be created by implementing in each class a static function that calls *new*. This static member function can be called when a particular type is needed.

In MFC, this member function is called *Construct()*. It is created by the IMPLEMENT_DYNCREATE or IMPLEMENT_SERIAL macros. One of these macros must appear exactly once in a .cpp module for each class supporting dynamic creation. In Scribble, the *IMPLEMENT_DYNCREATE (CScribDoc, CDocument)* statement appears near the top of SCRIBDOC.CPP. The first argument is the class, and the second is the class's parent class. Listing Two is the code generated by the preprocessor.

When MFC needs a document or any other *CObject*-derived type, it calls the *CreateObject()* member function for an instance of a *CRuntimeClass*. *CreateObject()* allocates the memory using the size in the *CRuntimeClass* structure, then calls *ConstructObject()*. *ConstructObject* verifies that the type supports dynamic construction, then calls the *Construct()* function.

The type registry is a linked list of CRuntimeClass *structures*

Although no explanation is given in the sources, this arrangement cleanly separates the construction of an object from the memory allocation. This seems like a lot of extra work, but it is required under certain circumstances. For example, if an array were created manually, the memory would have to be created with a single *malloc()* in order to be contiguous. By using *ConstructObject()*, you could manually initialize each array entry. This mechanism allows decisions normally made in C++ at compile time to be made at run time.

Example 2 shows the *Construct()* function. The syntax of the call to *new* is a little unusual. The function actually called is *CObject::operator new(size_t, void*)*. Remember, the size of a structure is an implied argument when operator *new* is called, but must be explicitly declared in the operator *new* definition. This version of *new* in *CObject* does nothing, but calling *new* has the side effect of calling the constructor for this object. Again, the memory was already allocated by *CreateObject* using the size information in *CRuntimeClass*.

By using the registry of *CRuntimeClasses* and the *Construct()* member function, MFC is able to lookup and create new types on the fly, which solves Problem 1. A potentially serious problem with this technique is that multiple inheritance and virtual base classes are not supported (see MFC Technical Note #16).

Type Registration

Problem 2 is that users must be able to easily add new classes into the registry. The idea of types registering themselves is core to object-oriented design. If a type registers its own existence with a registry instead of hardcoding the type into the registry, then the type can be freely added and removed from the program without any code changes to the registry.

Although not immediately obvious, it is the IMPLEMENT_DYNCREATE macro that enables users to easily add new classes into the registry. In the expansion of IMPLEMENT_DYNCREATE in Listing Two, the static instance of *CRuntimeClass* in *CScribDoc* is initialized as in Example 3.

Several of these entries have already been discussed. In particular, the statement *sizeof(CScribDoc)* is used by *CreateObject* to allocate memory; then the function pointed at by *CScribDoc::Construct* initializes the memory.

The next line places this information into the MFC type registry: *static const AFX_CLASSINIT _init_CScribDoc(&CScribDoc::classCScribDoc);*. This declaration constructs an object of type AFX_CLASSINIT using a constructor that takes a *CRuntimeClass* as its argument. Because this object is at file scope, it will be constructed before *main()*. AFX_CLASSINIT's constructor links this *CRuntimeClass* into the MFC type registry. AFX_CLASSINIT itself has no member data, so it will not take up any data space.

```
(a) class CScribDoc : public CDocument
    {
    protected: // create from serialization only
        CScribDoc();
        DECLARE_DYNCREATE(CScribDoc)
     ...
    };

(b) protected:
        static CRuntimeClass* __stdcall _GetBaseClass();
    public:
        static  CRuntimeClass classCScribDoc;

       virtual CRuntimeClass* GetRuntimeClass() const;
        static void __stdcall Construct(void* p);
```

Example 1: *(a) The beginning of a class declaration; (b) after preprocessing, the DECLARE_DYNCREATE macro expands to several new class members.*

```
void __stdcall CScribDoc::Construct
                              (void* p)
{
    new(p) CScribDoc;
}
```

Example 2: *The* Construct( ) *function.*

```
CRuntimeClass CScribDoc::
            classCScribDoc = {
    "CScribDoc",
     sizeof(CScribDoc),
    0xFFFF,
    CScribDoc::Construct,
    &CScribDoc::_GetBaseClass,
    0 };
```

Example 3: *Initializing the static instance of* CRuntimeClass *in* CScribDoc.

(continued from page 64)

This mechanism allows on-the-fly registration of object types whenever they are linked into the program, thus solving Problem 2.

A common question is, What is the difference between the various DECLARE and IMPLEMENT macros? All DECLARE_DYNAMIC and IMPLEMENT_DYNAMIC macros define a static instance of *CRuntimeClass* similar to the DYNCREATE shown earlier, except that the *Construct* field is NULL. DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE add the *Construct()* function for dynamic type creation. DECLARE_SERIAL and IMPLEMENT_SERIAL build on the DYNCREATE macros, and replace the 0xFFFF entry with the structure's schema number.

The SERIAL macros also define *operator>>* for the class. The *operator>>* requires special handling because it is called with a pointer to the class, but no instance of the class will exist until the instance has been serialized in from the file. Without an instance of the class, MFC cannot access the run-time class information to ensure that the object being loaded is equivalent to or is a derived class of the given pointer. By overloading *operator>>*, MFC is able to pass in a pointer to the run-time type information so that the serialization mechanism will be typesafe.

Creating Types from a File

The third problem is to create a mapping scheme to allow a type to be created based on information read from a file. Given that the compiler requires class names to be unique and that the class name is already embedded in the *CRuntimeClass* structure, the actual class name is the ideal candidate to write to a file in order to identify a class.

An object could be saved to an archive by writing the class's name and data. MFC does this and a little bit more for each class it encounters. The name of the class is from the structure's *CRuntimeClass*, which is obtained from a virtual function in the object. Because the typing is dynamic and done at run time, a structure of type *Tiger* will be properly written even if MFC is passed a pointer to a base class of type *Animal*. This type safety is important. Any function can safely save an object to an archive even if the object's exact type is unknown.

The same benefit applies to restoring an object from an archive. Going back to the example at the beginning of the article, MFC is able to successfully load the correct document type from a file with the simple statement in Example 4.

In the implementation of *operator>>*, MFC loads the name of the class from the file, then searches the list of types for that name. As long as the type exists in the registry and was declared with either DECLARE_DYNCREATE or DECLARE_SERIAL, MFC can construct the object. The actual loading of the data appropriate to that kind of object is delegated to the object itself by calling its *Serialize()* virtual member function, and Problem 3 is solved.

The type of object created is separated from the type of object requested. If a derived class is loaded into a pointer to a base class, as in the example of *CDocument*, then the correct derived class will still be created. This is the only way that the correct *vtbl* pointer can be set up to point at the object's virtual functions. If the object in the archive is not a "kind of" the specified object, MFC will throw an exception.

There are two potential pitfalls here for developers. First, renaming a serializable structure or class will corrupt any old save files or archives. Second, MFC does not record the length of each object into the archive. If MFC can't load an object, it will not be able to skip the object and load the rest of the archive.

Optimizing Archives

When I first scanned this code, I had visions of massive bloat in the save file and painful delays while the linked list was walked repeatedly. But MFC keeps its archive size down and its execution speed up by using hashed identifiers.

MFC keeps a hash table that tracks all classes and objects written to an archive. Once they are written, MFC does not rewrite them; it writes an identifier instead. Thus, when the archive is read back in, the linked list of types is traversed only for a new class. For subsequent instances of the class, the proper instance of *CRuntimeClass* will be found with a hashed lookup.

This behavior also means that multiple references to the same object are handled correctly. If objects A and B both point to a single instance of object C when the archive is created, they will both point to a single instance of object C when the file is read back in. MFC will also correctly resolve circular references between objects.

The implementation is much faster than I expected. On a 486/66, MFC was able to save and load an archive over a megabyte long with 10,000 separate instances of *CArray<DWORD,DWORD>* in under two seconds.

An important limitation is that the hash table can have no more than 32,766 classes and objects per archive context. This count only includes classes derived from *CObject* and serialized with *operator<<*, not fundamental types such as *short* and *long*, *CString*, and *CPoint*. (See *MFC Technical Note 2: Persistent Object Data Format* for more information on how archives are constructed.)

```
CDocument* pDoc;
CArchive& ar;
  ...
ar >> pDoc;
```

Example 4: *Loading the correct document type from a file.*

Schema Versi ns

A poorly documented feature in 32-bit MFC 3.2 is support for versionable schemas, where MFC allows the *Serialize()* routine to handle the various versions of the class instead of throwing an exception. This feature is very important in an evolving project. Although I will describe how to implement a versionable schema, the implementation is broken in Visual C++ 2.x and fails at run time. I recommend telling Microsoft how much you would like this feature fixed.

In MFC, each structure that uses DECLARE_SERIAL and IMPLEMENT_SERIAL has an associated version number. This number is normally set to 1, as shown in most MFC sample code; for example, *IMPLEMENT_SERIAL(CStroke, CObject, 1).*

Each structure or class has its own version number, which can vary independently of the others. MFC automatically writes the version number into the archive after the class ID. Prior to 3.0, MFC did not have a mechanism to completely support this version number, so older versions throw an exception if the schema number of the object in the file does not match the current schema number. This inhibits support for multiple schemas.

In MFC 3.0 and later, this behavior is only the default, and can be changed. By ORing the third parameter of the IMPLEMENT_SERIAL macro with the constant VERSIONABLE_SCHEMA, MFC will allow you to handle the schema version in your *Serialize()* function. For example, to set the document version number to 3, specify *DECLARE_SERIAL(CScribDoc, CDocument, VERSIONABLE_SCHEMA | 3).*

To use this feature, a class should call *GetObjectSchema()* in its *Serialize()* member function when the archive is loaded; see Listing Three.

C nclusion

By creating a foundation of a run-time type mechanism with a class factory and building serialization on top of it, MFC implements a fast, flexible, typesafe serialization mechanism. This mechanism should be powerful enough to satisfy most design requirements.

DDJ

(Listings begin on page 122.)

(continued from page 120)

```
            return NULL;
        }
    } else {
        return NULL;
    }
}
BOOL __export KMetaClass::CheckNextStrict( KFiler &f )
{
    ASSERT( k_attached );
    return KMetaClass::LoadNext( f ) == this;
}
BOOL __export KMetaClass::IsSubClass( char *name )
{
    // Relationship is 'receiver IsSubClassOf name'
    if ( name == m_name ) {
        return TRUE;
    } else if ( m_super_class != NULL ) {
        return m_super_class->IsSubClass( name );
    } else {
        TRACE( "KMetaClass::IsSubClass - %s is not a sub-class of"
               " %s.\n", m_name, name );
        return FALSE;
    }
}
KMetaClass * __export KMetaClass::Find( char *name, BOOL search_alias )
{
    KMetaClass      *k  = k_class_list;
    KMetaClassAlias *a  = k_alias_list;
    BOOL            f   = FALSE;
    //ASSERT( k_attached );
    while ( k != NULL && !f ) {
        if ( k->m_name == name || strcmp( k->m_name, name ) == 0 ) {
            f   = TRUE;
        } else {
            k   = k->m_next_class;
        }
    }
    if ( k == NULL && search_alias ) {
        // Search aliases.
        while ( a != NULL && !f ) {
            if ( strcmp( name, a->m_alias_name ) == 0 ) {
                k = Find( a->m_class_name, FALSE );
                f   = TRUE;
            } else {
                a = a->m_next_alias;
            }
        }
    }
    #ifdef _DEBUG
        if ( k == NULL ) {
            if ( search_alias ) {
                TRACE( "KMetaClass::Find - Failed to find class:"
                       " %s in class list or in alias list\n", name );
            } else {
                TRACE( "KMetaClass::Find - Failed to find class:"
                       "%s in class list\n", name );
            }
        }
    #endif
    return k;
}
void __export KMetaClass::AttachSubClass( KMetaClass *kc )
{
    kc->AttachSibling( m_sub_class );
    kc->m_super_class   = this;
    m_sub_class     = kc;
}
void __export KMetaClass::AttachSibling( KMetaClass *kc )
{
    m_sibling_class     = kc;
}
/* Meta class alias code. */
__export KMetaClassAlias::KMetaClassAlias( char *class_name, char *alias_name )
{
    m_alias_name    = alias_name;
    m_class_name    = class_name;
    m_next_alias    = k_alias_list;
    k_alias_list    = this;
}
__export KMetaClassAlias::~KMetaClassAlias( void )
{
}
```


MFC

Listing One

```
struct CRuntimeClass
{
// Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema;         // schema number of the loaded class
    void (PASCAL* m_pfnConstruct)(void* p); // NULL => abstract class
    CRuntimeClass* m_pBaseClass;

// Operations
    CObject* CreateObject();
// Implementation
    BOOL ConstructObject(void* pThis);
    void Store(CArchive& ar);
    static CRuntimeClass* PASCAL Load(
                CArchive& ar, UINT* pwSchemaNum);
    // CRuntimeClass objects linked together in simple list
    CRuntimeClass* m_pNextClass;// linked list of registered classes
};
```

Listing Two

```
void __stdcall CScribDoc::Construct(void* p)
{
    new(p) CScribDoc;
}
CRuntimeClass* __stdcall CScribDoc::_GetBaseClass()
{
    return (&CDocument::classCDocument);
}
CRuntimeClass CScribDoc::classCScribDoc = {
    "CScribDoc",
     sizeof(CScribDoc),
    0xFFFF,
    CScribDoc::Construct,
    &CScribDoc::_GetBaseClass, 0 };
static const AFX_CLASSINIT _init_CScribDoc(&CScribDoc::classCScribDoc);
CRuntimeClass* CScribDoc::GetRuntimeClass() const
{
    return &CScribDoc::classCScribDoc;
}
```

Listing Three

```
class CSmallObject : public CObject {
   DECLARE_SERIAL(CSmallObject);
   DWORD m_value;   // used to be unsigned short in version 1
};
IMPLEMENT_SERIAL(CSmallObject, CObject, VERSIONABLE_SCHEMA| 2);
CSmallObject::Serialize(CArchive& ar)
{
   if (ar.IsStoring()) {
      ...
   }
   else {
      DWORD nVersion = ar.GetObjectSchema();
      switch (nVersion) {

         case -1:
            // -1 shows that the structure was created with DYNCREATE, not
            // with SERIAL.  Seeing this value is probably an error.
            break;
         case 1:    // This version used unsigned short
            unsigned short oldval;
            ar >> oldval;
            m_value = oldval;
            break;
```

```
        case 2:
           // Current version uses DWORD
           ar >> m_value;
           break;
        default:
           // Bogus value - probably corrupt data file.
           break;
     }
   }
}
```

ENVIRONMENT VARIABLES

Listing One

```
/* envtst.c : This prints out the current envp[] set and the current Windows
   default environment settings. I used this to determine if a change to the
   environment settings had taken place. Build this as a Borland EasyWin
   (Windows command line) executable. 5/95 Fritz Lowrey
*/

#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) {
    int i;
    char    *denv;  /* default environment */

    printf("Program environment array:\n");
    for (i=0; envp[i] != NULL; i++)
        printf("%s\n", envp[i]);
    printf("\nWindows default environment:\n");
    denv = GetDOSEnvironment();
    while (*denv != NULL) {
        printf("%s\n", denv);
        denv += strlen(denv) + 1;   /* move to the next string */
    }
    exit(0);
}
```

Listing Two

```
/* winenv.c:  Build using Borland C++ EasyWin environment to allow for stdio
 function calls. Copyright John "Fritz" Lowrey, 24 May, 1995. This code and
 research that made it possible were done in conjunction with the University of
 Southern California University Computer Services Dept.
*/

#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>

#define DEMO

/* static variable for environment manipulation, not visible to other modules*/
static char     *lpNewEnv; /* pointer to environment space */
static HGLOBAL  hNewEnv;   /* handle to environment memory */
static int  ENVSIZE;       /* size of environment space */

/* LOADPARMS stucture needed by LoadModule */
struct LOADPARMS{
     WORD      segEnv;         /* child environment  */
     LPSTR     lpszCmdLine;    /* child command tail */
     UINT FAR* lpShow;         /* how to show child  */
     UINT FAR* lpReserved;     /* must be NULL       */
} ;
/* Initialize the environment space, ENVSIZE is the size of the environment
region (defined on the SHELL line of CONFIG.SYS */
/* returns -1 on error or 0 on sucess */
int EnvInit(int esize) {
    ENVSIZE = esize;
    if((hNewEnv = GlobalAlloc(GPTR | GMEM_SHARE, ENVSIZE)) == NULL)
        return -1;
    if ((lpNewEnv = GlobalLock(hNewEnv)) == NULL)
        return -1;
    /* we now have a pointer to the memory, fill it from the env space */
    if (memcpy(lpNewEnv, GetDOSEnvironment(), ENVSIZE) == NULL)
        return -1;
    /* environment space is initialized, return 0 */
    return 0;
}
/* definitions for new getenv and putenv routines */

/* Simple new getenv() routine.  Seach must be a label only */
LPSTR   NewGetEnv(LPSTR search) {
    LPSTR   tmpstr;

    /* point tmpstr at the environment space */
    tmpstr = lpNewEnv;
    /* scan through the space */
    while (tmpstr[0] != NULL) {
        /* if "search" is found at begining of tmpstr, return tmpstr */
        if (strstr(tmpstr, search) == tmpstr)
            return tmpstr;
        tmpstr += strlen(tmpstr) + 1;   /* move to next string */
    }
    /* if we fall through to here, return NULL */
    return NULL;
}
/* new putenv(): returns 0 on sucess -1 on failure */
int NewPutEnv(LPSTR putstr) {
    LPSTR   currentloc; /* currentlocation in the buffer */
    LPSTR   tmpstr;         /* used to move through buffer */
    char    label[30];  /* the lable portion of putstr */
```

(continued on page 124)